Project 3: Memory-System Optimization (10%)

ENEE 446: Digital Computer Design, Spring 2018

Assigned: Monday, Mar 26; Due: Monday, Apr 9 (+ pres. Wed Apr 11)

Purpose

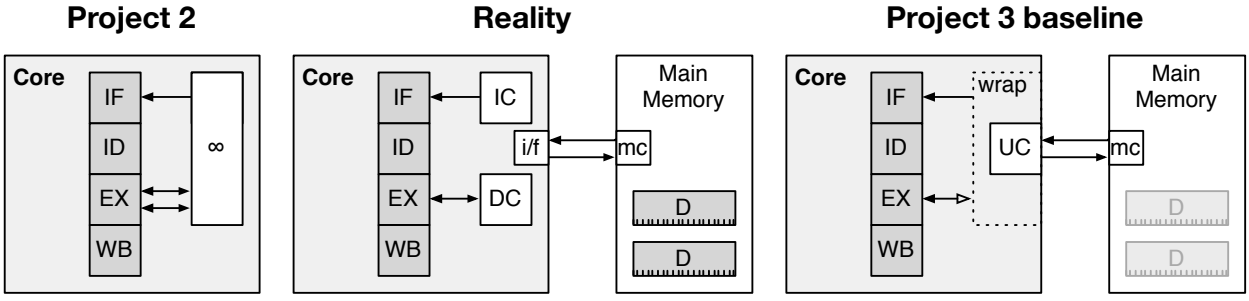
This project is intended to help you understand exactly what people mean when they say "the memory system is the problem." Your job is to get a cache system up and running (the cache, the cache controller, and the memory controller have all been developed for you; you need only integrate them into your P2 pipeline), measure its performance vs. your original P2 pipeline (i.e., quantify the overhead of imposing the realistic constraints of having to deal with a slow main memory system), and then improve it in any dimension that you feel like, either functionality or efficiency (performance), or any other dimension that you propose. You have to quantify the measurement, and it must be "real."

Project Overview

The figure below indicates what this is all about: in Project 2 (as well as in P1), we treated memory as being infinitely fast, infinitely large, and right next to the processor. Your processor could access memory as many times per clock cycle as you wanted and pull out as much data as you wanted, because we were focusing on other issues instead, so that you could learn the real implementation details behind *processor design*, including parallel instruction execution, vector processing, and pipelining. In this project, we will start focusing on the real implementation details behind *memory-system design*.

To begin with, main memory is not near, it is far away, and the pipe to it is narrow. A typical simple processor core has the ability to make two memory references per cycle: one in the fetch stage and one (not two) in the memory stage. These accesses are *not* to main memory but are instead to local caches, an instruction cache and a data cache. Each of these can provide, for example, a 64-bit word every clock cycle. In a 2GHz processor, that amounts to 32GB/s. When a reference misses a cache, a request goes to the bus interface unit, which sends the request to the memory controller handling main memory. Reading data from DRAM typically takes tens of nanoseconds, and the round trip can often be in the hundreds of nanoseconds (see graphs on that later). The memory channel between the processor and the memory controller is often a few tens of GB/s and is *far* less than the total cache bandwidth inside the processor (imagine if you had a dozen cores, each wanting 32GB/s; now imagine hundreds).

For Project 3, you start with your P2 solution and modify it to have an interface to memory, which will produce the *Project 3 baseline*, shown in the figure below, on the right. You have been given a unified cache (UC) that can process one request at a time (a traditional cache organization), as well as a bit of glue logic wrapped around it that gives the *appearance* of it being dual-ported. The cache has an integral bus interface unit, and so when it receives a request that it cannot handle, it automatically fetches the requested block from main memory. All of this has been done already, and you need only to integrate



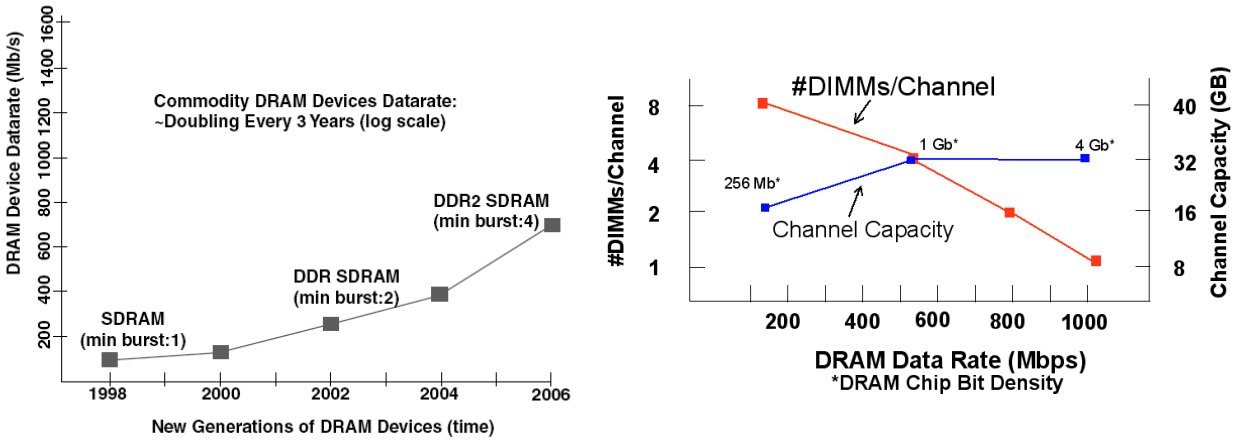


Figure 1. Trends showing datarate scaling over time (left), and channel capacity as a function of channel datarate (right). Figure on left taken from [Jacob et al. 2007]; figure on right taken from [Ganesh 2007].

the provided code and get it working. Your system will have the RiSC32 core as well as an instantiation of main memory, as shown in the "baseline" illustration. The code that integrates the two modules has been provided in the *test.v* file. Your task is first to measure this system against your P2 solution to see what sort of overhead a realistic memory system imposes, and then to improve the system, either by adding functionality or by making it more efficient and thereby improving its performance. To do this, you will first need to understand how the system works and what some of its primary limitations and inefficiencies are.

A Bit of Background

The memory system has become extremely important in recent years—memory is slow, and this is the primary reason that computers don't run significantly faster than they do. In large-scale computer installations such as the building-sized systems powering Google.com, Amazon.com, and the financial sector, memory is often the largest dollar cost as well as the largest consumer of energy. Consequently, improvements in the memory system can have significant impact on the real world, improving power and energy, performance, and/or dollar cost.

Performance Perspective—The Problem in Detail

The three main problem areas of the DRAM system today are its latency, its capacity, and its power dissipation. Bandwidth has largely been addressed by increasing datarates at an aggressive pace from generation to generation (see the left side of Figure 1) and by ganging multiple channels together. That is not to say bandwidth is a solved problem, as many people and applications could easily make use of 10–100x the bandwidth they currently have. The main point is that, in the past decade, the primary issue addressed at the DRAM-system level has been bandwidth: while per-device capacity has grown, per-system capacity has remained relatively flat; while bandwidth-related overheads have reduced, latency-related overheads have remained constant; while processor power has hovered in the 100W range, DRAM-system power has increased, and in some cases can exceed the power dissipation of the processor.

Power we will leave to the end of the chapter; one of the most visible problems in the DRAM system is its performance. As mentioned, bandwidth has been addressed to some extent, but main-memory latency, as expressed in processor clock cycles, has been increasing over time, i.e. getting worse. This was outlined famously by Wulf & McKee [Wulf & McKee 1995] and termed "the memory wall." Each generation of DRAM succeeds in improving bandwidth-related overhead, but the latency-related overhead remains relatively constant [Cuppu et al. 1999]: Figures 2(b) and 2(c) show execution time broken down into portions that are eliminated through higher bandwidth and those that are eliminated

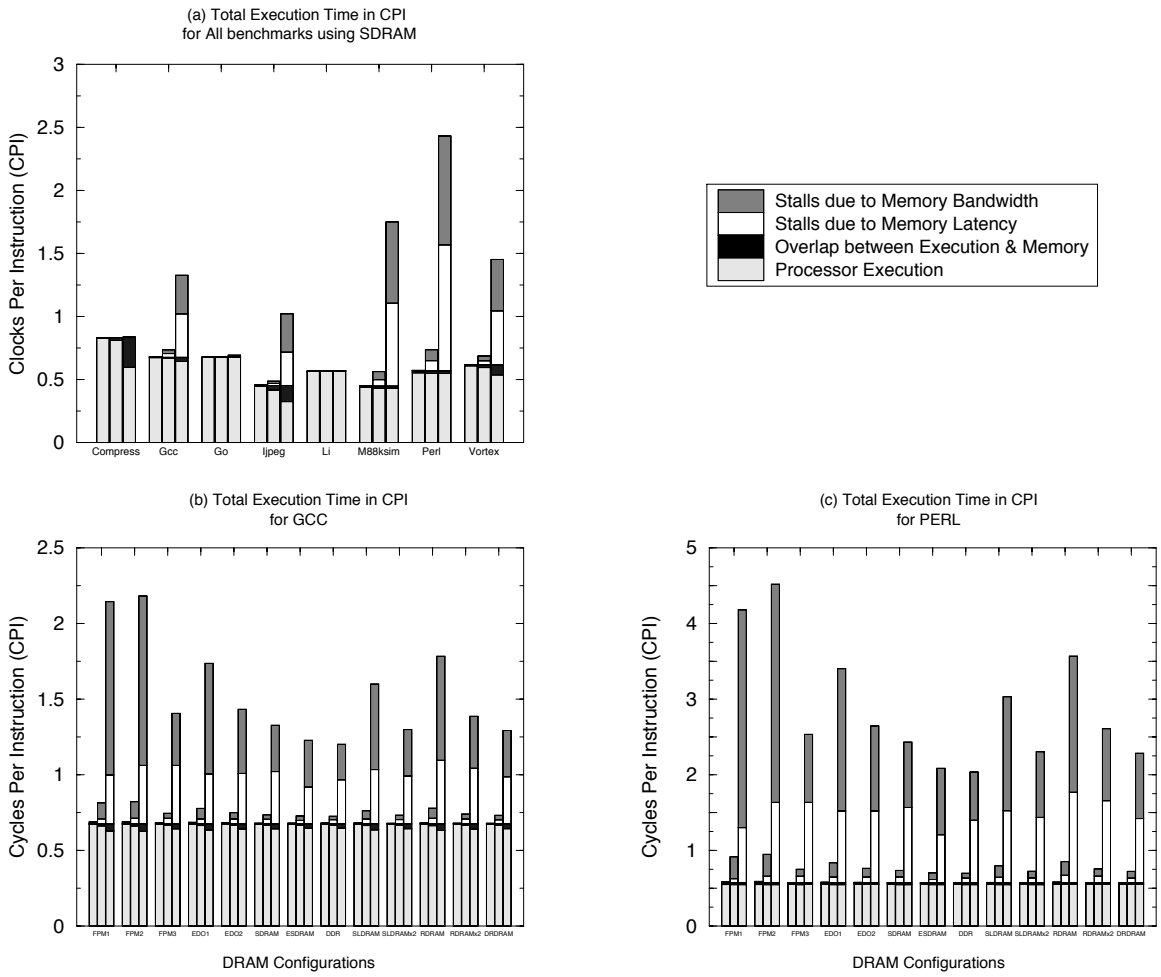


Figure 2. Total execution time including access time to the primary memory system. Figure (a) shows execution time in CPI for all benchmarks, using Synchronous DRAM. Figures (b) and (c) give total execution time in units of CPI for different DRAM types. The overhead is broken into processor time and memory time, with overlap between the two shown, and memory cycles are divided into those due to limited bandwidth and those due to latency. Figure taken from [Cuppu et al. 2001].

only through lower latency. When following succeeding generations of DRAM (e.g., when moving from Fast Page Mode [FPM] to EDO to SDRAM to DDR), one can see that each generation successfully reduces that component of execution time dependent on memory bandwidth compared to the previous generation—i.e., each DRAM generation gets better at providing bandwidth. However, the latency component (the light bars) remains roughly constant over time, for nearly all DRAM architectures.

Several decades ago, DRAM latencies were in the single-digit processor-cycle range; now they are in the hundreds of nanoseconds while coupled to processors that cycle several times per nanosecond and can process several instructions per cycle. A typical DRAM access is equivalent to roughly 1000 instructions processed by the CPU. Figure 3 shows access-latency distributions for two example benchmark programs (ART and AMMP, both in the SPEC suite): latencies are giving on the x-axis, in nanoseconds, and number of instances is given on the y-axis. An interesting point to note is the degree to which clever scheduling can improve latency—for instance the dramatic improvement for ART by moving from a simple FIFO scheme to the Wang algorithm—but as the AMMP results show (in which the latencies of many requests are reduced, but at the expense of significantly increasing the latencies of other requests), such gains are very benchmark-dependent.

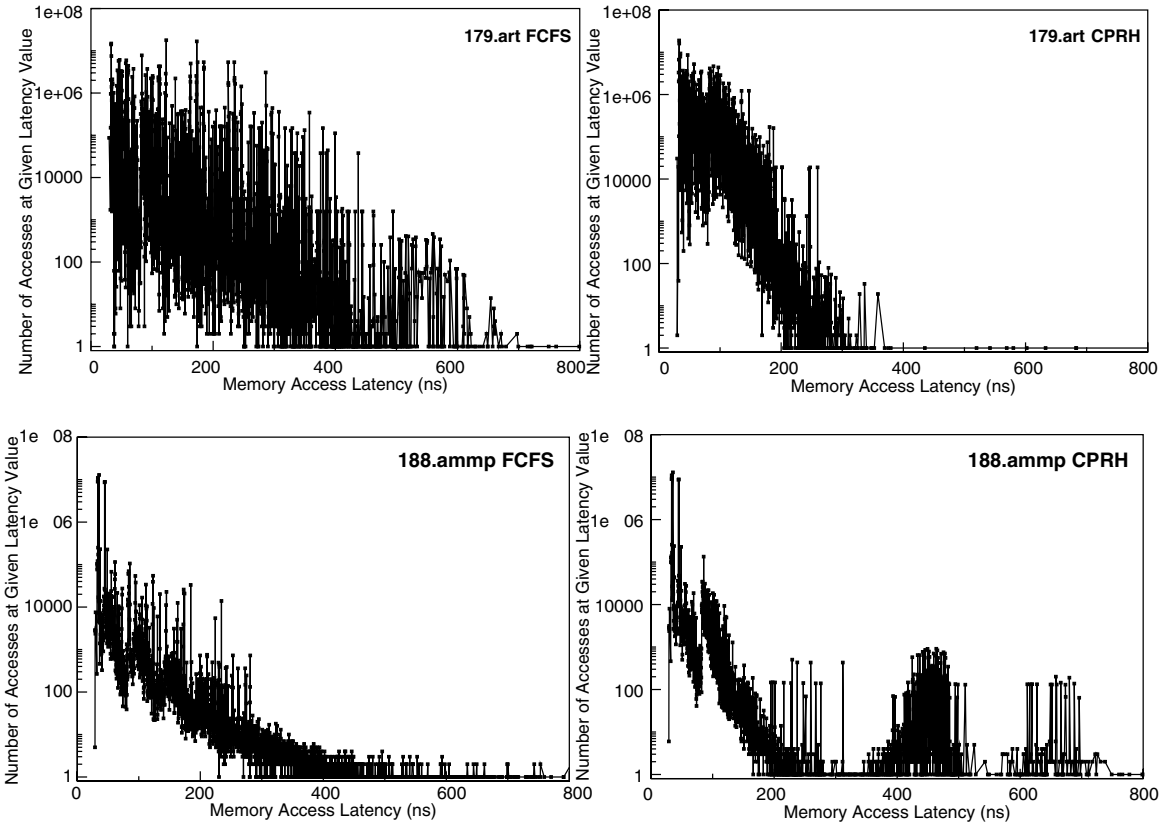


Figure 3. Range of access latencies for ART (one of the SPEC benchmarks), under two different scheduling heuristics: first-come, first-served (left) and Wang scheduling (CPRH, right). Figure taken from [Jacob et al. 2007].

Note that numerous techniques exist in the CPU to *tolerate* DRAM latency: these are mechanisms designed specifically to keep the CPU busy in the face of a cache miss, during which the processor would otherwise be stalled for many cycles waiting for the DRAM system. Examples include lockup-free caches, out-of-order scheduling, multi-threading, and prefetching. As should be intuitively obvious, these enable the CPU to get work done while waiting for data to return from main memory; they tend to provide this benefit at the cost of increased power dissipation and a need for additional memory bandwidth.

Nonetheless, the most effective mechanism found to date that enables us to overcome a slow back-end memory system is a fast *cache* at the front end.

Caches & Data Alignment

The textbook goes into great detail on the structure of caches, and you were doubtlessly introduced to them in your *Organization* class, so we don't dwell too much here on the fundamentals. This document tries only to bring out things the textbook doesn't cover, and which are important to this project.

One aspect is that of *data alignment*. Figure 4 illustrates the main concept: you have been able to think about main memory as a sequence of words up to this point, and your SIMD vectors have been able to start at any random location. This model does not work with caches, because the cache reads and writes data a *block* or *line* at a time, and that is nearly always a large number of *words* (typically, main memory is byte-addressed, and cache blocks are 32 bytes, 64 bytes, or larger). What this means in a practical sense is that you need to "pad" your code so that vectors always start on an address that == 0 mod 4. An example of this can be found in the *vcode1.s* file in the project directory. As an aside, all modern

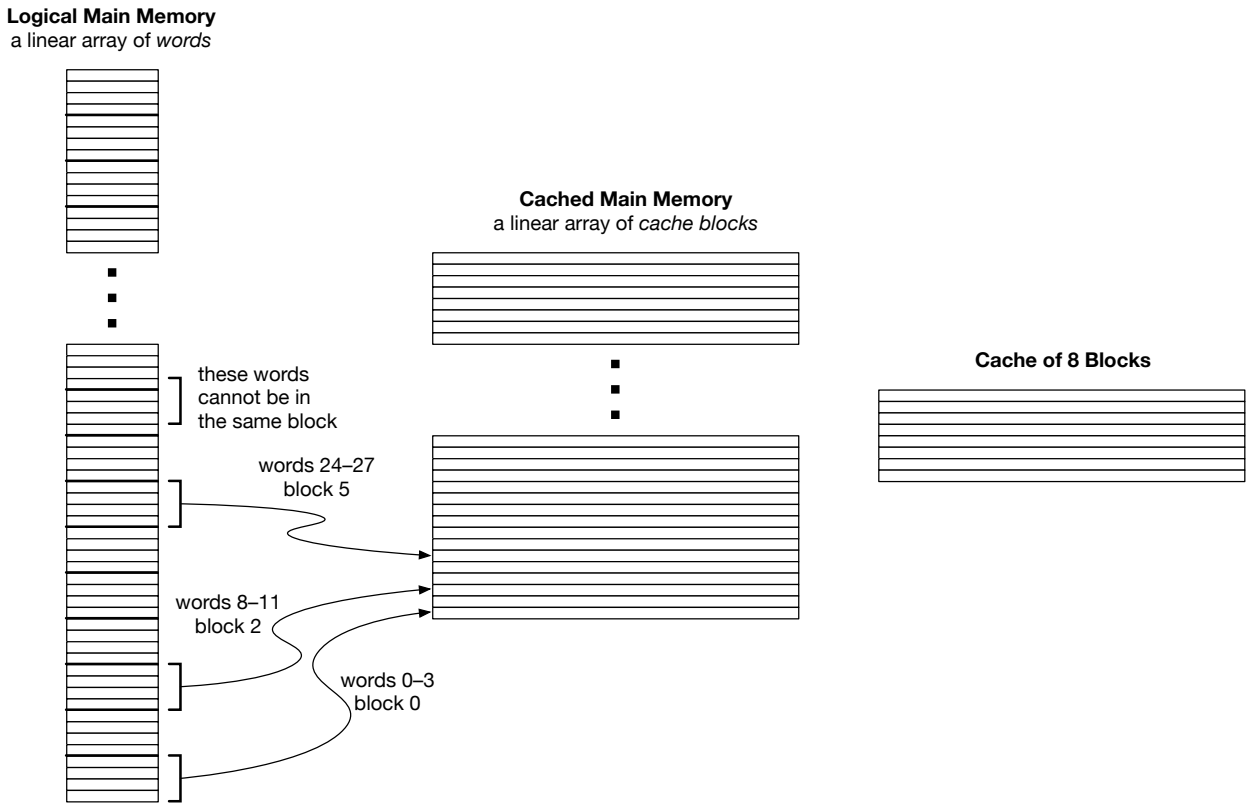


Figure 4. Main memory is typically introduced as a linear sequence of address locations … but that is just its logical structure. The reality, when caches are introduced, is that main memory must be thought of as a linear sequence of block-sized structures, because the cache block is the fundamental unit of transfer between the cache and main memory. Thus, if you have a data structure, such as a SIMD vector, that overlaps cache blocks, it will take two memory requests to fetch, and you will not be able to access it all at once.

compilers do this sort of thing automatically: data structures are always padded to align on cache-block boundaries.

Some of the Hairy Details

[excerpts taken from *The Memory System: You Can't Avoid It; You Can't Ignore It; You Can't Fake It*]

A further note on alignment: it can cause incredibly significant issues with performance. As you will see when you run your Verilog code, every time the required data is not in the cache, the entire processor stalls, waiting for the request to come back from the memory system. Imagine if your data layout causes problems you didn't intend.

When dealing with the cache as well as the with DRAM system, knowledge of your address patterns can help tremendously with application debugging and performance optimization, because different access patterns yield different latencies and different sustainable bandwidths.

Both caches and DRAMs have strong relationships with powers-of-two dimensions: turning memory addresses into resource identifiers (which bank, which row, which column, etc.) is much easier if the numbers and sizes of the resources are all powers of two, because that means one can simply grab a subset of the binary address bits to use as an identifier. Otherwise, one would have to do multiple passes of modulo arithmetic on every single cache or main-memory access. The power-of-two thing is important because software designers often like to use powers of two to define the sizes of their data structures, and this coincidence can ruin performance. As an example, consider the following code:

```
#define ARRAY_SIZE (256 * 1024)

int A[ARRAY_SIZE];
int B[ARRAY_SIZE];
int C[ARRAY_SIZE];

for (i=0; i<ARRAY_SIZE; i++)
        A[i] = B[i] + C[i];
```

Assuming that "int" maps to a 4-byte data quantity, each of these arrays is 1MB in size. Because they are defined by the programmer one after another in the code, the compiler will arrange them contiguously in (virtual) memory. Therefore, if array A is located at the address 0x12340000, then array B will be found at location 0x1244000, and C will be found at 0x12540000.

This addressing is a critical issue when trying to access the items in those arrays. When the C code is executed on a general-purpose processor, the processor will first look inside its caches to see if the data is there. Assume the first-level cache is direct-mapped 64KB with 16-byte cache blocks; for the first iteration of the loop (accessing the first 4-byte words of each array), the hardware starts with a read to item B[0], takes address 0x12440000 and uses the address-subset 0x000 to index into the cache—this is the set of 12 bits between the '4' of the address and the trailing '0'. That's all that is used to determine where in the cache the controller will look for the data, and if the data is brought in from main memory, that is where that block will be placed in the cache. The hardware does the same for item C[0], so it uses the address 0x000 to find the datum. When the processor writes datum A[0] to the cache, address 0x000 is used as well. All three requests to item 0 if arrays A, B, and C all go to the same spot in the cache. Since the cache is direct-mapped, it cannot hold all of these requests, so all will miss the cache.

It gets worse.

16 bytes were fetched into the cache to satisfy each of these 4-byte requests, which means that, in a normal cache scenario, we should only have to bring in one data-fetch to satisfy three following requests. That is the main point of organizing caches into large blocks: when we move sequentially through memory, the following requests should "hit" in the cache, even if the first request misses. However, because each successive request to each different array overwrites the same cache block, when the processor gets around to the second iteration of the loop, in which it accesses datum B[1], that datum will not be found in the cache, and instead A[0] will be found there, which will cause the hardware to fetch B[1]. Furthermore, A[0] is recently-written data, and this data will have to be written out before B[1] can be fetched.

The end result: in the first four iterations of the loop, the block containing B[0], B[1], B[2], and B[3] is fetched into the cache four times. The block containing C[0], C[1], C[2], and C[3] is fetched into the cache four times. The block containing A[0], A[1], A[2], and A[3] is fetched into the cache four times and then written out to memory four times. In an ideal cache, there would have been three reads (or, in some caches, just two) and one write. In our case, there was *four times* that amount of memory traffic.

Note that you can see this first-hand by playing around with the *vcode1.s* file and change the sizes of the two arrays to make them overlap, or not (as the code is written right now, there is no overlap).

The problem gets a bit better out at the L2 (level 2) cache, because further out the caches are often *set associative*. This means that a cache is able to place several different cache blocks at the same location, not just one. So a four-way set-associative cache would not fall prey to the problems above, but a two-way cache would.

Once we move to the off-chip L3 cache, all addresses are certainly physical addresses, instead of virtual ones*. The operating system manages virtual-to-physical mappings at a 4K granularity, and for all

* L2 caches in most processors today are still on-chip and, though they are typically physically indexed, can be virtually indexed.

intents & purposes in the steady state the mappings are effectively random (for more detail on virtual memory, see [Jacob & Mudge 1998a, 1998b, and 1998c] and the Virtual Memory chapter in [Jacob et al. 2007]). The compiler works in the virtual space, but (for general-purpose machines) at compile time it has no way of knowing the corresponding physical address of any given virtual address; in short, the compiler's notion of where things are in relationship to one another no longer holds true at the L3 cache. So, for instance, datum A[0] is almost certainly not located exactly 1MB away from datum B[0] in the physical space, just as datum A[1024] is almost certainly not located exactly 4KB away from A[0] in the physical space.

The end result is that this changes the rules significantly. Virtual memory, in the steady state (i.e., after the operating system has allocated, freed, and reallocated most of the pages in the memory system a few times, so that the free list is no longer ordered), effectively randomizes the location of code and data. While this undoes all of the compiler's careful work of ensuring non-conflicting placement of code and data (e.g., see [Jacob et al. 2007], chapter 3), it also tends to smear out requests to the caches and DRAM system, so that you reduce the types of resource conflicts described above that happen at the L1 cache—in particular, at both the L3 cache level and the DRAM level (the DRAM's analogue to cache blocks is an open row of data, and these can be many kilobytes in size), the code snippet above would not experience the types of pathological problems seen at the L1 level.

However, plenty of other codes *do* cause problems. Many embedded systems as well as high-performance (supercomputer-class) installations, use direct physical addressing or have an operating system that simply maps the physical address equal to the virtual address, which amounts to the same thing. Sometimes operating systems, for performance reasons, will try to maintain "page-coloring" schemes [Taylor et al. 1990; Kessler & Hill 1992] that map virtual pages to only a matching subset of available physical pages, so that the virtual page number and the physical page number are equal to each other, modulo some chosen granularity.

More commonly, algorithms access data in ways that effectively convert what should be a sequential access pattern into a *de facto* strided access pattern. Consider the following code snippet:

```
struct polygon {
        float   x, y, z;
        int value;
} pArray[MAX_POLYGONS];

for (i=0; i<MAX_POLYGONS; i++) {
        pArray[i].value = transform(pArray[i].value);
}
```

This walks sequentially through a dynamic list of data records—sequential access pattern, right? Effectively, no: the distance from each access to the next is actually a *stride*: a non-zero amount of space lies between each successively accessed datum. In this example, the code skips right over the x, y, and z coordinates and only accesses the values of each polygon. This means that, for every 16 bytes loaded into the processor, only 4 get used. We chose the struct organization for convenience of thinking about the problem and perhaps convenience of code-writing, but we're back to the pathological-behavior case. Better to do the following:

```
struct polygon {
        struct coordinates {
                float x, y, z;
        } coordinate[MAX_POLYGONS];
        int value[MAX_POLYGONS];
} Poly;

for (i=0; i<MAX_POLYGONS; i++) {
        Poly.value[i] = transform(Poly.value[i]);
}
```

This is a well-known transformation, called *turning an array of structs into a struct of arrays*. The name comes from the fact that, in the general case, e.g., if there are no groups of data items that are always accessed together, one might do something like the following:

```
struct polygon {
        float   xArray[MAX_POLYGONS];
        float   yArray[MAX_POLYGONS];
        float   zArray[MAX_POLYGONS];
        int vArray[MAX_POLYGONS];
} Poly;
```

This would make sense if one were to perform operations on single coordinates instead of the x,y,z triplet of each polygon.

The notion works even for dynamic data structures such as lists and trees. These data structures facilitate rapid searches through large quantities of (changing) data, but merging the indices with the record data can cause significant inefficiencies. Consider the following code:

```
struct record {
        struct record *left;
        struct record *right;
        char key[KEYSIZE];
        /* ... many bytes, perhaps KB, of record data ... */
} *rootptr = NULL;

struct record *findrec(char *searchKey)
{
        for (struct record *rp = rootptr; rp != NULL; ) {
                int result = strncmp(rp->key, searchKey, KEYSIZE);
                if (result == 0) {
                        return rp;
                } else if (result < 0) {
                        rp = rp->left;
                } else {        // result > 0
                        rp = rp->right;
                }
        }
        /* if we fall out of loop, it's not in the tree */
        return (struct record *)NULL;
}
```

Even though this might not seem amenable to optimization, especially for large databases, it can be. If the record size is on the order of a kilobyte (not unreasonable), then each record structure would occupy a kilobyte, and it only takes 1000 records to reach 1MB, which is significantly larger than an L1 cache. Any database larger than 1000 records will fail to fit in the L2 cache; anything larger than 10,000 records will fail to fit in even an 8MB L3 cache. However, what if we do the following to the data:

```
struct recData {
        /* ... many bytes, perhaps KB, of record data ... */
};

struct recIndex {
        struct recIndex *left;
        struct recIndex *right;
        char key[KEYSIZE];
        struct recData *datap;
} *rootptr = NULL;
```

If the *key* value is small, say 4 bytes, then the *recIndex* structure is only 16 bytes total. This means that one could fit the entire binary index tree for a 2000-record database into a 32KB level 1 cache. An 8MB L3 cache could hold the entire index structure for a half-million-record database. It scales beyond

the cache system: this technique of partitioning index from data is common practice for the design of very large databases, in which the indices are designed to fit entirely within main memory, and the record data is held on disk.

Lastly, another way the strided-v-sequential access issue manifests itself is in the implementation of nested loops. Writing code is deceptively simple: the coder is usually working very hard at the intellectual problem of making an algorithm that works correctly. What is not immediately obvious is that the structure of the code frequently implies an access pattern of the data. Consider the following two code snippets:

```
int pixelData[ HEIGHT * WIDTH ];

for (i=0; i<HEIGHT; i++) {
        for (j=0; j<WIDTH; j++) {
                compute( &pixelData[i * WIDTH + j] );
        }
}

for (j=0; j<WIDTH; j++) {
        for (i=0; i<HEIGHT; i++) {
                compute( &pixelData[i * WIDTH + j] );
        }
}
```

The only difference is that the loops have been exchanged. Assuming no hidden consequences from the *compute()* function, both code snippets should do exactly the same thing. And because both loops iterate one step at a time, it is tempting to think that both code snippets walk through memory sequentially, one integer at a time. However, while the first snippet walks through the *pixelData* array one integer at a time, the second snippet walks through the *pixelData* array touching one integer out of every WIDTH integers. For example, if the image size is 2000x3000 pixels, then the access pattern (in array indices) of the first code snippet is the following:

```
0, 1, 2, 3, …
```

This is sequential. It is predictable, and it uses every byte fetched from the memory system. By contrast, the access pattern of the second code snippet is the following:

```
0, 3000, 6000, 9000, …, 5994000, 5997000, 1, 3001, 6001, …
```

It should be clear that, while predictable, this sequence is hardly sequential, and it only uses one integer out of every cache block fetched.

Project Details

Your job is to implement a cache and main memory system in your P2 pipeline, and then, once you have it working, characterize it and improve it. You will work in teams, and you should use a project design that received all "perfect!" results on the autograder.

In the project directory there is a set of files, about half of which are the same as previously, and about half of which are either new or changed from Project 2.

In particular, two of the new files are *cache.v* and *memory.v*, where you will find the cache and main-memory structures. The *cache_wrapper* module should be added to your risc32 pipeline, and it simply replaces the *SRAM* module, keeping all of the original *MEM_xxx* wires intact—see the *PIPELINE.diff* file for the changes; very few modifications are required. The biggest changes are in the risc32 interface to the outside world: it now communicates with a main memory system over a 64-bit bus, and so the *test.v* file needs to instantiate both CPU and memory, and connect the two.

Cache Implementation & Limitations

The cache that has been provided to you is extremely simple. It has 8 blocks and a single port. Each block is the size of one 128-bit (4-word) vectors. The *cache_wrapper* that is implemented around it tries to make its one port look like two, so that the cache can satisfy two "simultaneous" requests from the pipeline: one from instruction fetch, and one from a data operation in the execute stage. It does this by delaying and buffering, which obviously adds overhead, and it isn't simultaneous access at all.

Additional limitations:

1. The *cache_wrapper* cannot handle two data requests at the same time, thus instructions of the form

   ```
   lw r2, r3, 4 | lw r5, r6, 7
   ```

   are not allowed, unless you modify your system to handle it. As an example, I have modified the code in the *vcode.s* file for you so that it adheres to this limitation; a version that only does one load at a time is in the *vcode1.s* file.

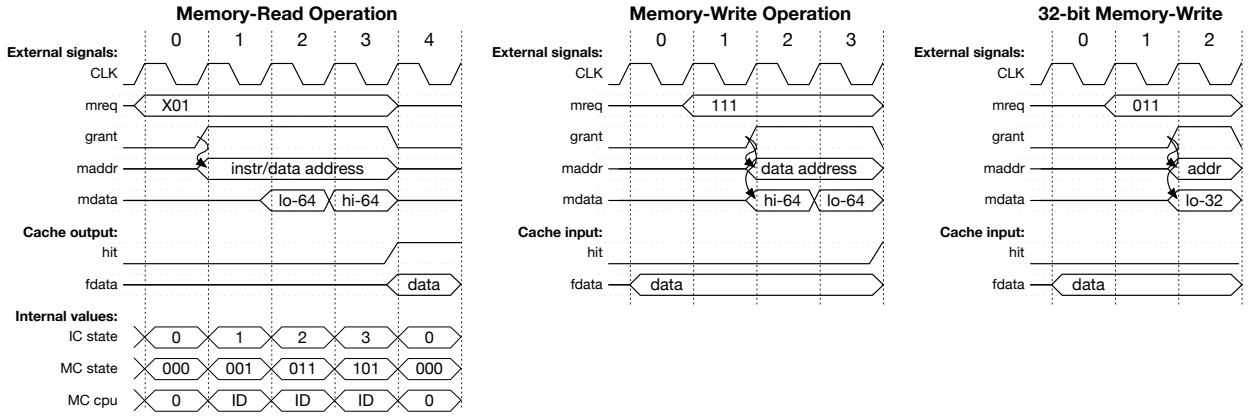
2. The *cache_wrapper* cannot handle stores properly. The Cache module *can* handle them and writes them through to main memory (it is a write-through cache), but the *cache_wrapper* does not. Thus, you cannot use SW or VSW instructions unless you modify your system to handle them.

What this means in practical terms is that code that works on your P2 pipeline will *not necessarily* work on your P2+cache pipeline. Again, take a look at the *vcode1.s* file.

Cache-Miss Operations & Timing

As mentioned, the cache block is 128 bits (4 words) long, and the interface to main memory is 64 bits. That means that it takes several cycles to transmit a single cache block. In modern systems, cache blocks are often 64 bytes, and the bus width is 64 bits. meaning that it takes 8 cycles to transmit one block. As you step through your code simulations, you should get a good idea of just how painful this is.

The protocol between our cache and main memory is shown below:



It is relatively straightforward; when a request is made to the cache, and the corresponding block is not in the cache, then the cache makes a request to main memory. It raises a request, *mreq*, which indicates the size of the request (128-bit or 32-bit), whether the request is read or write (note: all reads are treated by the memory system as 128-bit), and the validity of the request. When the memory controller sees this, it responds with a *grant* signal, and the CPU responds by sending out the desired data address, and, if the request is for a store operation (not supported by the cache_wrapper, but implemented in the cache), then the data as well. For the read operation, the internal states are also given in the figure.

Reminder: Data-Alignment Issues

One last reminder: if you try to use a VLW instruction that has an address with the bottom two bits not equal to '0' then you will get an unpleasant surprise. Go back and look at Figure 4 again.

Suggestions for Your Consideration re: Optimizations

Example improvements/optimizations you could perform:

1. Improve the protocol between the CPU and the memory controller to reduce the number of cycles required.
2. Improve the operation of the "cache wrapper" glue logic to reduce the number of cycles required.
3. Implement a *split-cache* structure, with separate I- and D-caches, to allow true simultaneous instruction fetch and data operations.
4. Add support for *three* simultaneous requests (e.g., two LW atoms in one instruction bundle, which need to be handled at the same time as an instruction fetch).
5. Add support for **store** operations.
6. Make the cache access lockup-free.
7. Change the associativity of the cache (it is currently direct-mapped).
8. Double the block size, which will require messing with the bus protocol.

Some of these examples improve functionality; others improve performance. Feel free to suggest a completely different type of improvement. However, the improvement should not be trivial, such as making a bus wider or increasing the number of blocks in the cache (you *can* increase the number of blocks; just don't count that as your sole improvement).

Some of you will note, especially after reading the *test.v* file, that the bus protocol and memory-controller design are set up for multicore. That is Project 4 (it rhymes), where you will look into multitasking and cache coherence. Feel free to explore optimizations that anticipate that direction.

Project Submission & Grading

You have three deliverables for this project:

1. Submit your working project as a tarball of all your files, including a README file in PDF that includes diagrams of what you have done. Your documentation need not be elaborate, but it must cover everything you did. Use the on-line *submit* facility and number this as assignment #3.
2. Do three performance characterizations (your graph/table can be included in the README file):
   a. Your P2 design
   b. Your initial P3 design (Project 3 baseline)
   c. Your improved P3 design

   The measurements should be done using a fairly substantial program as a benchmark … for instance, you can use the program in *vcode1.s*, but modify it to use longer vectors (the example on the website is short so that everything fits in the cache and is easy to analyze).
3. One person from your team will give a **5-minute presentation** in class on what you did, **Wednesday April 11th**. You will all grade each other on what your teams implemented as improvements. Your team's grade will be the average of the scores you receive from the rest of the class, and everyone on your team will receive the same grade.